Numpy

We have covered the basics of the Python language. We know how to work with the basic data types, create functions and handle exceptions. To continue learning Python, we will implement the ordinary least squares (OLS) estimator for the parameters of a linear regression under the classical assumptions.

Let's assume that the economic model of interest is given by:

$$\underbrace{y}_{n\times 1} = \underbrace{X}_{n\times K}\underbrace{\beta}_{K\times 1} + \underbrace{\varepsilon}_{n\times 1}$$

where y is a $n \times 1$ vector of observed dependent variables, X is an $n \times K$ matrix of observed explanatory variables, β is a $K \times 1$ vector of unknown parameters and ε is a $n \times 1$ vector of unobserved explanatory variables.

We know that the OLS estimator of β is given by:

$$\hat{\beta} = (X'X)^{-1}X'y$$

Our first objective is to implement this estimator.

Using a bottom-up approach, we need to understand:

1. The basic data types in Python;
2. How to create vectors and matrices;
3. How to do matrix transposition, inversion and multiplication;
4. How to create a function that given X and y produces $\hat{\beta}$;
5. How to generate data to test out our code.

Step 1 was already accomplished with our last lecture. We now turn to creating vectors and matrices in Python.

To do so, we will use a package called NumPy. NumPy is a package for scientific computing in Python. It implements an object to represent N-dimensional arrays (vectors, matrices and higher dimensional matrices). It also has linear algebra functions and random number generators.

Why do we need Numpy? Python is a dynamically typed language, it infers the type of a variable at runtime. This means that when Python store variables in memory, it not only stores the variable's value, but also its type. Then, when we perform a computation, like adding two variables (`x+y`), Python looks up the type of `x` and the type of `y`, and applies the definition of `+` to those types if it makes sense to do so. If those types are integers, for example, then Python performs an integer addition. However, if the type of `x` is integer, but `y` is a list, then the operation is not defined and Python will raise an exception (error). This type of checking is called a runtime type check.

The runtime type check makes programming in Python a pleasure, since you do not have to worry about types all the time. However, it does lead to inefficiencies when we start performing operations on large datasets. Numpy solves those inefficiencies.

Numpy introduces a new type of list, called a Numpy array. In an array, the type of all elements is the same. This has two benefits. First, the type is only stored once, so performing operations with numpy is much faster than with Python lists. Second, since all elements have the same type, their position in the memory is easy to compute. This means that accessing random elements in a numpy array is quick.

Numpy also provides a set of operations for numpy array, all implemented in C. Arrays are not required to be 1 dimensional, so the class can also deal with matrices and higher dimensional arrays. Numpy is the base of other higher-level packages, like `pandas` and `scipy`.

1 Install

Numpy can be installed with `pip`. After activating your environment in the terminal, run:

```
pip install numpy
```

Now `numpy` is available for Python. In the jupyter notebook, we can import the package by executing:

```
import numpy
```

The `import` will look for the package named `numpy`, will find it and load all of its content. The methods, variables and objects defined in the Numpy package will be available in the object `numpy`.

You will often see the following command being used:

```
import numpy as np
```

Which makes the `numpy` functionality available via `np`. This is often used because it requires less typing to use numpy.

2 Basics

To create a matrix with numpy we run:

```
matrix = np.array([[0, 1, 2, 3],
                   [4, 5, 6, 7]])
```

This creates a new matrix with dimensions 2 by 4.

Basic properties of `np.ndarray`:

```
type(matrix)
print(f'Shape of matrix = {matrix.shape}')
print(f'Number of axes = {matrix.ndim}')
print(f'Total number of elements = {matrix.size}')
```

The elements of the matrix can be accessed by using its indices. Since the matrix has 2 dimensions, we need to give it two indices: an index for the row and an index for the column. Remember that in Python the indexing starts at 0.

```
print(matrix[0, 0])                    # element at row 0 and column 0
print(matrix[1, 3])                    # last element of the 2nd row
print(matrix[1, -1])                   # last element of the 2nd row
print(matrix[-1, -1])                  # element at last row and last colu
```

We can use : to slice the matrix and obtain all values of a certain row or of a certain column:

```
print(matrix[0, :])              # first row, all columns
print(matrix[:, 0])              # all rows, first column
print(matrix[0, 1:3])            # first row, columns 1 and 2
print(matrix[:, 1:3])            # all rows, columns 1 and 2
```

We can also select specific columns or rows:

```
print(matrix[:,[0, 3]])          # all rows, first column and last column
```

Like the built-in function `range`, numpy also provides a way to generate a sequence of numbers with the function `np.arange`:

```
vector = np.arange(20)
```

We can reshape this vector into a matrix:

```
matrix = vector.reshape(4,5)
```

Notice that when converting a vector into a matrix we need to decide whether to put the values first into columns, or first into rows. For example, if we have a vector $[1, 2, 3, 4]$ and want to reshape it into a $2x2$ matrix, there are 2 natural possibilities: create the new matrix by filling each row first, or create the matrix by filling each column first.

$$\text{Fill Rows: } \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \text{ vs. Fill Columns: } \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

The default in Python is to fill each row before moving to the next (first matrix above). The behavior is controlled by a keyword argument `order` given to the reshape method.

```
a = np.arange(4)
# Different ways of reshaping
# fill each row, row by row
a.reshape((2, 2), order='C') # this is also the default
# fill each column, column by column
a.reshape((2, 2), order='F')
```

We can also get the type of the values stored in the matrix:

```
print(matrix.dtype)
```

Observe that the `np.array` takes a list of numbers as input, it does not take numbers as separate inputs:

```
np.array([1,2,3,4])              # correct, creates a vector
np.array(1,2,3,4)                # wrong
```

If a list of lists is given, then we get a two-dimensional array:

```
np.array([[0, 1], [1, 2]])
```

3 Data Types

The data type of values stored in a matrix are usually inferred when the values are assigned:

```
matrixOfIntegers = np.array([[0, 1, 2, 3, 4, 5],
                             [5, 4, 3, 2, 1, 0],
                             [10, 9, 8, 7, 6, 5]])
print(matrixOfIntegers.dtype)
matrixOfFloats = np.array([[1.0, 2.3],
                           [-5.1, 9.8234]])
print(matrixOfFloats.dtype)
```

The first matrix has the type `int64` and the second matrix has the type `float64`. Notice that if we try to change a value in the first matrix to a float, it will be implicitly converted to an integer:

```
print(matrixOfIntegers[0, 0])
matrixOfIntegers[0, 0] = 0.34
print(matrixOfIntegers[0, 0])
```

We can specify the type of data when we first create the matrix:

```
matrix = np.array([[0, 1], [2, 3]], dtype='float64')
matrix[0, 0] = 3.2
print(matrix)
```

The matrix is initialized with integers, but we specify the type to be a float, so the values are implicitly converted to floats.

The most common data types for numbers are:

Data Type	Description
int_	default integer type (usually int64)
float_	default float type (usually float64)
complex_	default complex type (usually complex128)
bool_	default boolean type (uses a byte)
int<bytes>	where <bytes> can be 8, 16, 32 or 64
uint<bytes>	unsigned integer
float<bytes>	
complex<bytes>	

Remember that defining a data type will convert the initial values to the defined type:

```
matrix = np.array([[0, 1],
                   ['False', True],
                   ['a', 'b']], dtype='bool_')
```

If the values cannot be converted, an error is raised:

```
matrix = np.array([[0, 1], [2, 'abc']], dtype='float_')
```

The command raises a `ValueError` and informs us that the string 'abc' cannot be converted to a float.

It is also possible to store strings in numpy arrays (matrices):

```
names = np.array(['A', 'B', 'C', 'D'], dtype='str_')
print(names.dtype)
```

Notice that the type is actually "<U1". The "U" stands for "unicode string" and "1" is the number of characters. Since all values passed to the `np.array` constructor have just 1 character, numpy assumes all data that we will store on this matrix will have a single character. This might not be the case, and if we try to store a bigger string:

```
names[0] = 'Guilherme'
print(names[0])
```

Instead of storing the entire string, only the first character was stored. We can let numpy know that we need more space in memory by defining how many characters we need:

```
names = np.array(['A', 'B', 'C', 'D'], dtype='<U100')
names[0] = 'Guilherme'
print(names[0])
```

Now we can store 100 characters in each element of the matrix.

Numpy can handle other data types, even objects:

```
grades = np.array([{'A': 9, 'B': 8, 'C': 10},
                   {'A': 'Econ', 'B': 'Finance', 'C': 'Econ'}])
```

In this case the data type is `object`, and `grades` is a matrix with two elements, where each is a dictionary. The type `object` is used for creating arrays that can hold any data type, since everything is an object in Python. In this case, we are loosing the efficiency that comes with Numpy, since we are creating an array that can hold different types of data. In some cases we might still want to do so, since Numpy provides several useful functions for dealing with arrays.

The complete documentation on specifying data types (`dtype`) can be found here. Arrays can also hold more than one type, details on how to hanlde multiple types can be found here.

4 Empty and Pre-Filled Arrays

On many cases we will want to create an empty array and populate as the code is executed. We can create arrays pre-filled with ones or zeros via:

```
ones = np.ones((4, 5))
zeros = np.zeros((10, 3))
```

The functions `np.ones` and `np.zeros` take a tuple as input, which defines the shape of the matrix to be created.

The function `np.empty` creates an "empty" matrix of the given shape and data type:

```
empty = np.empty((4, 10), dtype='float_')
```

The matrix is filled with whatever values are in the memory. If you use this command make sure all values are correctly replaced, otherwise you may run into bugs.

It is also possible to pre-fill a matrix with any given value:

```
full = np.full((10, 3), 3.14)
manyStrings = np.full((5,5), 'StudentName')
```

5 Operations

Arithmetic operations with numpy arrays are always element wise.

```
a = np.arange(10)
b = np.ones(10)
print(f'Type a = {a.dtype}, b = {b.dtype}')
c = a + b
print(c)
print(f'Type c = {c.dtype}')    # implicit conversion of a to float
print(c*10)
print(c - 10)
print(c**2)
```

Notice that `**` is the same as `pow(c,2)`.

Comparisons are also done element wise:

```
print(c > 5)
```

Booleans can be used to recover elements of an array:

```
c[c>5]
```

You can recover the indices that satisfy a condition with the `np.where` function:

```
indices = np.where(c <= 2)
print(indices)
print(c[indices])
```

It is possible to manipulate all elements of an array and accumulate the values:

```
# some random values to stand in as returns
returns = np.random.random((78, 5))
returns /= 100                          # divides everything by 100
help(np.sum)
RV = np.sum(returns**2, axis=0)
annRV = 100*np.sqrt(252*RV)
print(f'RV = {RV}\n'
      f'annualized RV = {annRV}')
```

The `np.random.random` is a function that generates random values in the interval $[0, 1)$. The `np.sum` function accumulates over elements in an array by summing them. Calling `np.sum` without the `axis` argument will sum all of the elements in a matrix, resulting in a single value. The keyword argument `axis=0` instructs the sum to occur along the rows. Numpy provides many universal mathematical functions, such as `np.sqrt` for computing the square root of a number (remember, element wise).

We can find the minimum and maximum values in an array:

```
print(f'Min RV = {np.min(annRV)}\n'
      f'Max RV = {np.max(annRV)}')
print(f'Index of Min = {np.argmin(annRV)}\n'
      f'Index of Max = {np.argmax(annRV)}')
```

Numpy arrays also implement a multitude of different methods:

```
x = np.array([4,3,2,1]);
# in-place sort
x.sort()
```

```
print(x)
# common operations
print(x.sum(), x.mean(), x.max(), x.min())
# equivalent to
print(np.sum(x), np.mean(x), np.max(x), np.min(x))
# index of extremum
print(x.argmin(), x.argmax(), np.argmin(x), np.argmax(x))
# cummulative sum
x.cumsum()
# cummulative product
x.cumprod()
# variance
x.var()
x.std()
x.std() == x.var()**0.5
# search sorted array for index where a new value can be inserted
# without affecting the sort
x.searchsorted(2.5)
```

Numpy also implements matrix operations:

```
# Inner Product: <a, b>
x = np.array([1, 2, 3])
y = np.array([-1, 0, 10])
print(np.dot(x, y))
# Matrix multiplication
# create identity matrix
a = np.eye(2)
b = np.random.random((2, 2))
print(np.matmul(a, b))
# The symbol @ is overloaded for matrix multiplication
print(a @ b)
print(a @ b == np.matmul(a,b))
# Matrix transposition
print(np.transpose(b))
print(b.T)
# Matrix inversion
c = np.array([[1, 2],
              [3, 4]])
print(np.linalg.inv(c))
```

Numpy also implements other functions, like singular value decomposition (`np.linalg.svd`), eigenvalues (`np.linalg.eig`), Moore-Penrose inverse (`np.linalg.pinv`), Kronecker product (`np.kron`) and others. For more information on linear algebra with Numpy check the reference page.

6 Mutability and Copying Arrays

Create a new array:

```
x = np.array([4,3,2,1], dtype='float_')
# arrays are mutable
```

```
x[0] = 10.32
print(x)
```

What happens when we assign a new name to x?

```
a = x
a[1] = -2.34
# What is printed?
print(a, a[1], x[1])
```

Notice that `x` is a name to an array, and `a` is a reference to `x`. Thus, `a` is also a name to the exact same array.

```
a == x
a is x
```

This is a sensible behavior for memory efficiency. If you do need to make a copy of an array, then:

```
print(x)
a = np.copy(a)                    # deep copy
print(a)
np.all(a == x)
a is x
a[:] = np.pi
print(a, x)
```

We can check if all elements of an array are `True`:

```
(a == x).all()
np.all(a==x)
# built in
all(a == x)
```

7 Additional Functionality

Numpy has several functions that can be applied element-wise:

```
x = np.array([1, 2, 3, 4])
# vectorized functions (element-wise)
print(np.sin(x))                  # implicit type conversion
print(np.log(x), np.exp(x))
```

These functions are referred to as universal functions (`ufunc`). They are universal in the sense that they work with arrays (work with many elements).

We can compose these operations:

```
# +, -, /, * and ** also work element-wise
print(np.log(x + 1)*3/np.exp(x)**2)
```

If a function is not universal (also known as vectorized), applying it on an array will lead to error:

```
def f(x):
    return 1 if x > 0 else 0
f(x)
```

The issue here is that `bool(x)` is not defined for numpy arrays. While `x > 0` works, the part that says `if x > 0` is equivalent to `if bool(x > 0)`, which is undefined for arrays.

```
bool(x)
# Remember that for Python lists bool is always defined
bool(list(x))
```

We can vectorize a function:

```
f = np.vectorize(f)
f(x)
```

The function `np.vectorize` is basically creating a for-loop around `f`:

```
res = np.zeros(len(x))
for i in np.arange(len(x)):
    res[i] = f(x[i])
print(res)
```

We can accomplish the result of the function `f` using np.where:

```
help(np.where)
# returns 1s where x > 0, and 0 otherwise
np.where(x > 0, 1, 0)
```

Remember that comparisons are element-wise and can be used for slicing:

```
# comparisons are element-wise
a = np.array([-1, 0, 1])
b = np.array([1.2, -2.3, 3])
a > b                          # implicit type conversion for a
a > 0.2
# slice based on comparisons
a[a > 0.2]
a[a > b]
```

The Numpy subpackage `np.random` implements several random number generators. You can get more information about them with `help`:

```
help(np.random)
```

The Numpy subpackage `np.linalg` has other linear algebra tools. You can also use `help` to learn more about them:

```
# numpy subpackage np.linalg implements linear algebra functions
help(np.linalg)
```

A table that compares commands in Matlab to commands in Numpy is available here. A good overview of many numpy features is available here.

8 Implementing the OLS Estimator for Linear Regressions

We now know everything we need to know about Python to estimate parameters in a linear regression:

```
def linreg_ols(x, y):
    assert x.shape[0] == y.shape[0], "Different number of rows"
    beta = np.linalg.inv(x.T @ x) @ x.T @ y
    return beta
```

Generate some test data to test `linreg_ols`:

```
# 1000 data points with a constant + 3 explanatory variables
x = np.hstack((np.ones((1000, 1)), np.random.random((1000, 3))))
print(x)
beta = np.random.random((4, 1))
epsilon = np.random.random((1000, 1))
y = x @ beta + epsilon
print(y)
```

Estimate beta and compare the estimate to the true value:

```
beta_hat = linreg_ols(x, y)
print(np.hstack((beta, beta_hat)))
```

Let's update `linreg_ols` so that adding a constant to `x` is automatic:

```
def linreg_ols(x, y):
    assert x.shape[0] == y.shape[0], "Different number of rows"
    x = np.hstack((np.ones((x.shape[0], 1)), x))
    beta = np.linalg.inv(x.T @ x) @ x.T @ y
    return beta
```

Test it again:

```
x = x[:, 1:]
print(x)
print(x.shape, y.shape)
beta_hat = linreg_ols(x, y)
print(beta_hat)
```

Update `linreg_ols` so that adding a constant can be optionally specified by the user:

```
def linreg_ols(x, y, intercept=True):
    assert x.shape[0] == y.shape[0], "Different number of rows"
    if intercept:
        x = np.hstack((np.ones((x.shape[0], 1)), x))
    beta = np.linalg.inv(x.T @ x) @ x.T @ y
    return beta


# if no intercept is supplied, than a column of ones is added
print(linreg_ols(x, y))
print(linreg_ols(x, y, True))
print(linreg_ols(x, y, 1))
print(linreg_ols(x, y, False))
```

Always remember to add documentation to your functions (your future self will thank you):

```
def linreg_ols(x, y, intercept=True):
    """Estimates linear regression coefficients with OLS.

```

```
    Estimates beta in a linear regression: y = x @ beta + epsilon.
    Uses the ordinary least squares estimator.

    Args:
        x (np.array): A nxK matrix of explanatory variables
        y (np.array): A nx1 matrix of dependent variables
        intercept (bool): Specifies whether to estimate intercept coefficie

    Returns:
        beta (np.array): A Kx1 vector with the beta estimates

    Raises:
        AssertionError: x and y have different number of rows
        LinAlgError: if x is a singular matrix

    Example:
        linreg_ols(np.array([[1, -2], [0.5, -3.1]]), np.array([1, 2.3]))
    """
    assert x.shape[0] == y.shape[0], "Different number of rows"
    if intercept:
        x = np.hstack((np.ones((x.shape[0], 1)), x))
    beta = np.linalg.inv(x.T @ x) @ x.T @ y
    return beta


help(linreg_ols)
```

9 Save and Load Results

After running linear regressions, we might want to save the results. We can use `np.save` for that. It saves the results in a file for later use. The function `np.save` saves in a binary file, which is faster for loading compared to `.csv` files (but it is not human-readable).

```
beta = linreg_ols(np.random.random((1000, 3)),
                  np.random.random((1000, 1)))
print(beta)

help(np.save)
np.save('regression_estimates', beta, allow_pickle=False)  # npy extension
```

If you are only using the data only on your computer (not sharing), then you can set `allow_pickle=True` to speed things up.

To load the data, use `np.load`:

```
# load the data
del beta
print(beta)                          # not defined
beta = np.load('regression_estimates.npy', allow_pickle=False)
print(beta)
```

We can save the results in a `.csv` file, which is easier to share and inspect. To do so, we use `np.savetxt`:

```
np.savetxt('regression_estimates.csv', beta, delimiter=',')
# load from .csv
del beta
print(beta)
beta = np.loadtxt('regression_estimates.csv', delimiter=',')
print(beta)
```

We can also use np.loadtxt to load data created outside Python. Download this data set on housing prices in California, and save the file in your working folder. The data contains information on household prices in California in 1990 (based on the census at that time).

Look at the data file to understand it first. Skip first line since they are the headers. Ignore the last column since it is string data. So, we need to select all but the last column.

A missing value in a `.csv` file is represented as ,,. The value would be considered as an empty string (") by Numpy. The file has some missing data, so we also need to deal with it.

Load the data:

```
data = np.loadtxt('housing.csv', delimiter=",",
                  skiprows=0, usecols=range(9),
                  converters={4: lambda x: 0.})
# delimiter: to specify its a csv file
# skiprows: skips the first row, which has headlines
# usecols: to select first 9 columns
# converters: apply the anonymous functions to all strings it encounters
#             thus, transforming '' into a 0.0
print(data.shape)
print(data)
```

The function `np.loadtxt` is useful for files that are well formated and do not require much work. Later we will learn another package that implements a database object and provides much more powerful loading functions.

10 Organizing Outputs

Let's extend $\text{linreg}_{\text{ols}}$ so the R-squared is also computed:

```
def linreg_ols(x, y, intercept=True):
    assert x.shape[0] == y.shape[0], "Different number of rows"
    if intercept:
        x = np.hstack((np.ones((x.shape[0], 1)), x))
    beta = np.linalg.inv(x.T @ x) @ x.T @ y
    # residuals
    e = y - x @ beta
    rsquared = 1 - (np.sum(e**2)/np.sum((y-y.mean())**2))
    return (beta, rsquared)


# we are returning a tuple, where the first element are the estimated
# betas, and the second element is the R-squared
x = np.random.random((1000, 5))
y = np.random.random((1000, 1))
```

```
results = linreg_ols(x, y)
print(results)
print(results[0], results[1])
# tuple unpacking (a.k.a. unpacking):
beta, rsquared = results
beta, rsquared = linreg_ols(x, y)
```

As we start adding more outputs to a function, it can become harder to identify what is each output represents. We can organize the outputs by using a dictionary:

```
def linreg_ols(x, y, intercept=True):
    assert x.shape[0] == y.shape[0], "Different number of rows"
    if intercept:
        x = np.hstack((np.ones((x.shape[0], 1)), x))
    beta = np.linalg.inv(x.T @ x) @ x.T @ y
    # residuals
    e = y - x @ beta
    rsquared = 1 - (np.sum(e**2)/np.sum((y-y.mean())**2))
    return {'beta': beta, 'r2': rsquared}


results = linreg_ols(x, y)
# Check what are the outputs:
print(results.keys())
# Assign variable names
beta = results['beta']
rsquared = results['r2']
# A dictionary can also be unpacked:
beta, rsquared = results.values()
print(beta, rsquared)
```

We can also organize the output using a `namedtuple`. A `namedtuple` is similar to a tuple, but its elements have names. And the elements of a `namedtuple` can be accessed with the `.` notation:

The `namedtuple` is defined in the `collections` package, which is built-in Python. The `namedtuple` is a factory function, which returns a subclass. A subclass is what defines an object, it is just a definition. But, we can use it to create instances of the object, which can actually be used. `namedtuple` takes the name of the object as an input, and also a list of properties that the object will have.

```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
mouse = Point(0.3, 0.7)
print(mouse.x)
print(mouse.y)
```

We can adapt the example above to organize the output of `linreg_ols`:

```
def linreg_ols(x, y, intercept=True):
    assert x.shape[0] == y.shape[0], "Different number of rows"
    if intercept:
        x = np.hstack((np.ones((x.shape[0], 1)), x))
    beta = np.linalg.inv(x.T @ x) @ x.T @ y
```

```
    # residuals
    e = y - x @ beta
    rsquared = 1 - (np.sum(e**2)/np.sum((y-y.mean())**2))
    RegressionOutput = namedtuple('RegressionOutput', ['beta', 'rsquared'])
    return RegressionOutput(beta, rsquared)


results = linreg_ols(x, y)
# Check what are the field names:
print(results._fields)
# Recover the values
print(results.beta)
print(results.rsquared)
```

This is a more efficient way of providing results, since you can name the outputs. Also, the outputs are saved in a tuple, which is immutable and so is slightly more memory efficient.

11 Assignment

All assignments should be submitted to the Github repository you have been assigned to. The deadline is **August 12th by 11 PM**. You should write a report in Latex with the solutions to the problems below. If the problem requires you to code, then the code should also be included in the report. You can add code to Latex with the lstlisting package (see this stack post) or the minted package (see this other stack post).

Problem 1 *Read the* `numpy.random` *documentation page. Test the linear regression function using a normal distribution for the unobserved heterogeneity and some other distribution of your choice for the independent variables.*

Problem 2 *Extend* `linreg_ols` *to compute the standard error of the OLS estimates under the classical assumptions. Remember that under the classical assumptions, the standard error of the OLS estimates:*

$$SE(\hat{\beta}_i) = \sqrt{s^2 \cdot [(X'X)^{-1}]_{[i,i]}}$$

where s^2 *is computed from the residuals of the estimation:*

$$e \equiv y - X\hat{\beta}$$
$$s^2 \equiv \frac{e'e}{n-K}$$

Remember to update the output of the function and extend its documentation.

Problem 3 *Download this data set on housing prices in California, and save the file in your working folder. This data set was first used in Pace and Barry (1997), and was later modified for use in Géron (2017). The data contains information on household prices in California in 1990 (based on the census at that time). The first line of the file contains the name of the explanatory variables. Run the linear regression suggested in Equation (8) of Pace and Barry (1997).*

Problem 4 *Python has been around since 1990 and has a huge number of packages. Figure out if Python has a function or a package (or many packages) for estimating the parameters of a linear regression using OLS. If it does, what is the name of the function or the package? What are the outputs of the function you found?*

Problem 5 *When the classical assumption of homoskedasticity fails, we need a different estimators for the standard errors of the OLS estimates. White (1980) proposes a heteroskedasticity-robust estimator for the standard errors of the OLS estimates, which is now known as White's standard error. Equation 2.4.1 in Hayashi (2000) shows White's standard error:*

$$\widehat{SE(\hat{\beta}_i)} \equiv \sqrt{\frac{1}{n}[S_{xx}^{-1}\hat{S}S_{xx}^{-1}]_{[i,i]}}$$

There is a slight change of notation in this part of the Hayashi (2000), and x_i' is the ith row of X (x_i is a column vector with the explanatory variables in the ith row of X). The term S_{xx} is the sample mean of $x_i x_i'$: $S_{xx} = \frac{1}{n}\sum_{i=1}^{n} x_i x_i'$ (Equation 2.3.6 in Hayashi (2000)). The term $\hat{S}$ is an estimator for a matrix of fourth moments, and it is defined as $\hat{S} = \frac{1}{n}\sum_{i=1}^{n} e_i^2 x_i x_i'$ (Equation 2.5.1 in Hayashi (2000)), and e_i is as before (residual for the ith observation). Implement this estimator in the `linreg_ols` *function. You could write a local function to estimate White's standard error in the* `linreg_ols.m` *file.*

Problem 6 *(Optional)*

Implement the t-test based on White's standard error. Also, compute the p-value from given the statistic.

Problem 7 *Newey and West (1987) proposes another estimator to the standard errors for the OLS estimates under weaker assumptions. The estimator the authors propose is robust not only to heteroskedasticity, but also autocorrelation, and is known as the HAC standard errors. Does the package you found in Problem 4 have such a function to compute the HAC standard errors? If not, can you find some other package that does have it? If so, what is the name of the package?*

Problem 8 *(Optional)*

Implement the t-test based on the HAC standard error. Also, compute the p-value from given the statistic.

Problem 9 *Extend* `linreg_ols` *to accept an optional input named* `cov_type`*, which specifies the type of standard errors estimator to report. The variable* `cov_type` *can take one of many values, say "Standard", "White" or "HAC". Your function should be able to check if* `cov_type` *is one of those types and act accordingly. If it is not, the function should raise an appropriate error alerting the user with a very clear message of what is wrong.*

References

Géron, Aurélien (2017). *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.". URL: https://isbnsearch.org/isbn/9781491962299.

Hayashi, F. (2000). *Econometrics*. Princeton University Press. ISBN: 9780691010182. URL: https://books.google.com/books?id=QyIW8WUIyzcC.

Newey, Whitney K. and Kenneth D. West (1987). "A simple, positive semi-definite, heteroskedasticity and autocorrelation consistent covariance matrix". In: *Econometrica* 55.3, p. 703. URL: https://login.proxy.lib.duke.edu/login?url=https://search-proquest-com.proxy.lib.duke.edu/docview/214867593?accountid=10598.

Pace, R Kelley and Ronald Barry (1997). "Sparse spatial autoregressions". In: *Statistics & Probability Letters* 33.3, pp. 291–297. URL: https://doi.org/10.1016/S0167-7152(96)00140-X.

White, Halbert (1980). "A heteroskedasticity-consistent covariance matrix estimator and a direct test for heteroskedasticity". In: *Econometrica* 48.4, pp. 817–838. URL: https://www.jstor.org/stable/1912934.